Composition and Formal Validation in Reactive Adaptive Middleware

Annie Ressouche, Jean-Yves Tigli, Carillo Oscar

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Composition and Formal Validation in Reactive Adaptive Middleware

Annie Ressouche — Jean-Yves Tigli — Oscar Carrillo

N° 7541

February 2011

Perception, Cognition, Interaction

ISSN 0249-6399 ISRN INRIA/RR--7541--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SOPHIA ANTIPOLIS - MÉDITERRANÉE

Composition and Formal Validation in Reactive Adaptive Middleware

Annie Ressouche* , Jean-Yves Tigli† , Oscar Carrillo‡

Domain : Perception, Cognition, Interaction
Équipe-Projet Pulsar

Rapport de recherche n° 7541 — February 2011 — 27 pages

Abstract: Nowadays, adaptive middleware plays an important role in the design of applications in ubiquitous and ambient computing. Currently most of these systems manage the adaptation at the middleware intermediary layer. Dynamic adaptive middleware are then decomposed into two levels : a first one to simplify the development of distributed systems using devices, a second one to perform dynamic adaptations within the first level. In this report we consider component-based middleware and a corresponding compositional adaptation. Indeed, the composition often involves conflicts between concurrent adaptations. Thus we study how to maintain consistency of the application in spite of changes of critical components and conflicts that may appear when we compose some component assemblies. Relying on formal methods, we provide a well defined representation of component behaviors. In such a setting, model checking techniques are applied to ensure that concurrent access does not violate expected and acceptable behaviors of critical components.

Key-words: service oriented middleware, event-based composition, reliability, formal methods, synchronous modelling, validation

* Inria Sophia-Antipolis Méditerranée
† I3S Laboratory and CNRS (Rainbow)
‡ Inria Sophia-Antipolis Méditerranée

Centre de recherche INRIA Sophia Antipolis – Méditerranée
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Composition et vérification formelle dans les middlewares réactifs et adaptatifs

Résumé : De nos jours, les middlewares adaptatifs et réactifs jouent un role important dans la conception d'applications dans le domaine de l'informatique ubiquitaire et ambiante. Généralement, ces systèmes réalisent cette adaptation au niveau intermédiaire du middleware. Ainsi, les middlewares adaptatifs sont décomposés en deux parties : une première partie qui permet un développement simplifié des systèmes distribués utilisant des dispositifs, une seconde qui réalise les adaptations dynamiques de la première partie. Dans ce rapport nous considérons des middlewares à base de composants et une adaptation compositionnelle. Mais souvent lors d'une composition certaines adaptations concurrentes s'avèrent conflictuelles. Pour résoudre ce probème, nous étudions comment préserver la consistence d'une application lors de changements concernant certains composants critiques, avec des conflits qui peuvent apparaître quand on compose des assemblages de composants. Nous utilisons des méthodes formelles pour modéliser le comportement des composants afin de bénéficier des techniques de vérification par model checking et ainsi prouver que des accès concurrents respectent les comportements acceptables des composants critiques.

Mots-clés : middleware orientés services, composition par évènements, sûreté de fonctionnement, méthodes formelles, modèles synchrones, validation

Contents

1 Introduction

Ubiquitous computing follows an evolution of computer science introduced by Weiser [23] two decades ago. A major consequence is the arrival of applications more and more opened on every day environment relying on objects supposedly communicating and intelligent. Devices managed in ubiquitous computing are nowadays *heterogeneous*, *variable*, and *interacting with a physical environment*. Moreover, applications in this domain must often face some *variability* during execution time. Moving with a mobile user, such applications have not always access to the same devices. Thus, it turns out that the appearance and disappearance of these latter need a dynamic evolution of the application. Hence, evolving in a real environment, ubiquitous applications must be able to react to changes in the surrounding *physical environment*.

Then, it is a real challenge to address these constraints for middleware. Indeed, now they must support a stable applicative model in spite of a heterogeneous and variable software infrastructure. Actually, middleware must be

reactive (they must react to context change) and *adaptive* (they must adapt themselves continuously to context changes).

1.1 Component-based Adaptive and Reactive Middleware

Historically, [20] defines two extremes in the range of strategies for adaptation. At one extreme, adaptation is entirely the responsibility of individual applications. The other extreme of application-transparent adaptation places entire responsibility for adaptation on the system. Currently most of the work converge to manage the adaptation at the middleware intermediary layer [1]. In this last case dynamic adaptive middleware are then decomposed into two levels [7]. The primary level of middleware is to simplify the development of distributed systems [11]. The second level performs dynamic adaptations within middleware.

Because ubiquitous computing is based on preexisting devices, middleware must manage legacy of black-box of software pieces. Three kinds of approaches are well suited to manage such constraints : component oriented middleware, service oriented middleware and more recently new popular approaches using components assembly to compose preexisting services like, SCA or SLCA ([21]).

The second level of adaptive middleware manage dynamic modifications of the first one to perform adaptation. According to [15] we can distinguish two main approaches to implement software adaptation. The first one is *parameter adaptation*. For component based middleware this approach consists in modifying components variables that determine their behavior. The second one is *compositional adaptation*. This approach allows component-based middleware to change dynamically components with others in response to changes in its execution environment. In this paper we study how to maintain consistency of the application in spite of critical components changes and conflicts that may appear when we superpose component assemblies in mechanisms for compositional adaptation. Indeed in such cases, we need to use verification techniques to check safety and various other correctness properties of the evolving application.

1.2 Need for Validation

Then, the main motivation appears when we introduce new requirements for ubiquitous applications such as *safety*. Indeed, few research works in ubiquitous computing address some partly critical applications. For example, many ubiquitous applications address health care domain without validating some critical functionalities. Anyway, *safety* is an important concern in adaptive middleware. Applications may intervene in critical systems (i.e. system whose failure or malfunction may result in death or serious injury to people, or loss or severe damage to equipment or environmental harm). Components may have to satisfy stringent constraints related to security and should be submitted to formal verification and validation. Moreover, context change adaptation should preserve safety rules. Then key problems are : (1) how to specify and validate the behavior of one assembly connected to a critical component (Cf. section 3), (2) in case of multiple assemblies sharing some critical components, how to compose them and validate properties of the overall application (Cf. section 4).

1.3 Our proposal

The major contribution of this work is to show that formal methods (and particularly synchronous modelling framework) offer means to automatically validate critical component behaviors and to prove safety property preservation through a sound composition operation useful to perform context adaptation. Thus we extend our component-based adaptive middleware with specific tools to allow validation of local composition on critical devices and services, using model checking techniques.

The report is organized as follows: next section (2) briefly describes the component-based middleware use we consider and introduces the example we rely on all along the report to illustrate our approach. It is extracted from a use case in the domain of health care for elderly. Then section 3 presents our solution which introduces synchronous monitors to model critical devices expected behaviors. They support formal validation. In section 4 we introduce a composition operation between synchronous monitors preserving validated properties. Such an approach allows us to offer a deterministic solution to multiple access to critical devices. We discuss the practical issues of our work in section 5. We introduce our reactive adaptive middleware for ubiquitous computing, named WComp and also its extension with verification facilities. Then we describe the implementation of the example in our middleware. In section 6 we compare our approach with different works which address the problem of reliability of middleware for ubiquitous computing. Finally, section 7 concludes and opens the way for future works.

2 Component-based Middleware Use

In this work, we consider middleware where communication means are event-based. Of course event-driven systems are not suitable for very complex design, but adequate for reactivity, dynamicity and high adaptability. In our approach such components are often proxies for services for device and then must reflect the device behavior. Some of them are critical and we want to validate their usage within some middleware assemblies.

We illustrate our approach with the design of (a small part of) an application in the domain of health care for elderly. The purpose is to monitor old adult in an instrumented home, using sensing technology. There are different kinds of sensors in the environment: video cameras, contact sensors to indicate closed or opened status of equipment, wearable sensors, etc. In this framework, we are deep in the domain where reactive and adaptive middleware solutions apply, since some sensors can appear and disappear (particularly wearable ones). In this example, we show the design of a small part of a project dedicated to observe activities of daily living (ADLs) in an equipped home [1]. We consider the recognition of activities related to kitchen usage. The goal is to send several kinds of alarms depending on sensor observation results. Component proxies are associated to four sensors: a contact sensor on the fridge which indicates the state of the door (opened or closed); a timer which sends a minute information; a camera which locates a person; a posture sensor which tells if the person

[1] http://gerhome.cstb.fr/

is standing, sitting or lying. This latter is a wearable device composed by accelerometers.

In this application, an *Alarm* component proxy receives three kinds of alarms: *warning*, *weak_alarm* and *strong_alarm*. It is linked with assemblies of components for fridge and timer sensors, camera sensor and posture sensor. This *Alarm* component is critical and we will ensure that it raises the appropriate alarm in the designed application. To this aim, we offer a mean to ensure that each output event coming from one of the assemblies for sensors is correctly sent. Indeed, we supply a new component reflecting the behaviors of (assemblies of) components and checking these behaviors we can ensure that these components are used out of harm's way (see section 3.1.2). Moreover, it is not sufficient to individually prove that each new component outputs are not misconnected. We also must ensure that the combination of two output events coming from two different assemblies and linked with the same input event of *Alarm* component works correctly. Thus we introduce a safe composition between components (see section 4.1).

3 Components with Validated Behaviors

To validation purpose, we introduce models to describe the behavior of application components. Finite automata are well adapted to the representation of device behaviors and moreover provide a lot of verification tools based on efficient model-checking techniques to verify properties.

3.1 Component Behavior Modelling.

3.1.1 Component Behavior as Synchronous Models

The aim is to define means to represent component behavior. These components listen to events coming from other components or from an input environment and will provide output events in reaction. They have to satisfy stringent constraints (correctness, response time) and they should be submitted to formal verification and validation as they may intervene in a critical decision. Thus determinism would be an important advantage. A way of reducing the complexity of behavior description is to consider them evolve through successive phases. During one phase, only the external events which were present at the beginning of the phase and the internal events that occurred as a consequence of the first ones are considered. The phase ends when some stability (fixed-point) has been achieved (or when an external clock decides that it is over). We call such a phase an instant. Indeed, during such an instant, time seems to be suspended (the external events are frozen). Such an instant-based representation will be called a *synchronous* model. In such models, a reaction has *no duration* because its real duration is delayed to the next clock cycle or next instant of the system. This issue characterizes the *synchronous hypothesis* on which all synchronous models rely. A significant way well suited to validation is to express them as Mealy machines [16]. Mealy machines are both finite automata and synchronous models. Indeed a transition in Mealy machines corresponds to a reaction or an instant of the system.

The Mealy machines we consider are 5-uple of the shape:
$< Q, q^{init}, I, O, \mathcal{T}, \lambda >$. where Q is a finite set of states; $q^{init} \in Q$ is the initial

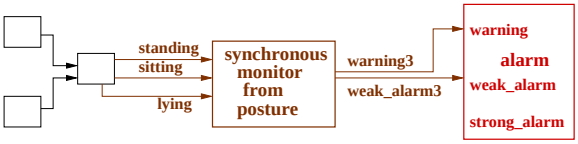

Figure 1: A synchronous monitor to manage the access to a critical alarm proxy component from an assembly of components related to a posture sensor.

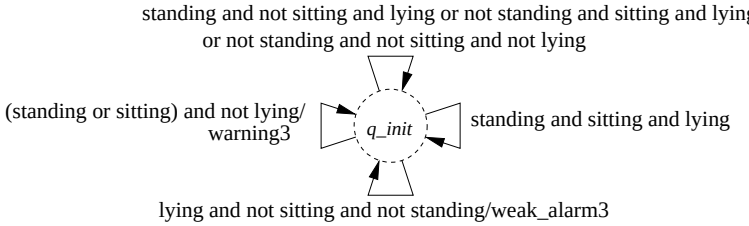

Figure 2: The posture monitor is a Mealy machine($< Q_3, q_3^{init}, I_3, O_3, \mathcal{T}_3, \lambda_3 >$) with an only state and two transitions.

state. I (resp. O) is a finite set of input (resp. output) events; $\mathcal{T} \subseteq Q \times Q$ is the transition relation. λ is a labeling function: $\lambda : \mathcal{T} \times I^B \mapsto 2^O \cup \{\epsilon\}$ where I^B is the set of Boolean expressions over I [2]. It is a Boolean algebra with standard interpretation for $true, false, \cdot, +$ and $\neg$ [3]. Finally, ϵ represents an undefined event [4]. In short, $q \xrightarrow{i/o} q'$ will denote a transition with the agreement: $(q, q') \in \mathcal{T}$ and $\lambda((q, q'), i) = o$. Furthermore, according to the synchronous hypothesis, we want our model deterministic and reactive:

1. $q \xrightarrow{i/o_1} q_1$ and $q \xrightarrow{i/o_2} q_2 \in \mathcal{T} \Rightarrow q_1 = q_2$ and $o_1 = o_2$ (determinism)
2. $\forall i \in I^B, \forall q \in Q, \exists q \xrightarrow{i/o} q' \in \mathcal{T}$ (reactivity)

3.1.2 Synchronous Monitors

Critical components will provide a synchronous model of their behavior and some additional properties (constraints) checked when component is used. This model is designed as a Mealy machine where each output is connected with an input event of the critical component. Indeed, let us consider a synchronous monitor specified as the Mealy machine $M =< Q, q^{init}, I, O, \mathcal{T}, \lambda >$ and connected to a critical component with I_C as input event set. There must exist an injective mapping : $in : O \mapsto I_C$.

Figure 1 illustrates such a situation. It shows a part of the application introduced in section 2. In this latter, there is an assembly related to the

[2] Its elements are built according to the following grammar: $e := true \mid false \mid I \mid e \cdot e \mid e + e \mid \neg e$.

[3] We will consider usual Boolean algebra rules to infer equality between two elements of I^B.

[4] For short, we will denote $X \cup \{\epsilon\}$ as X_ϵ.

posture sensor and connected to the *Alarm* component. Thus, we will define a synchronous monitor to describe the behavior of the assembly. This *posture* monitor listens to $I_3 = \{sitting, standing, lying\}$ input event set. Its output event set is $O_3 = \{warning_3, weak\_alarm_3\}$. It emits a $warning_3$ event when the person is sitting or standing and a $weak\_alarm_3$ event when he(she) is lying [5] . This monitor is detailed in figure 2.

Finally, the critical *Alarm* proxy component has $I_A = \{warning, weak\_alarm, strong\_alarm\}$ as input event set and there is an injection $in_3 : O_3 \mapsto I_A$: $\begin{cases} in_3(warning_3) = warning \\ in_3(weak\_alarm_3) = weak\_alarm \end{cases}$

A synchronous model becomes a monitor component beyond the unsafe proxy component. Then, safety and liveness properties concerning critical component usage can be verified using model-checking tools.

3.2 Component Behavior Validation

Among others validation techniques, the *model-checking* approach [5, 14] requires a model of systems against which formulas are checked for satisfaction. The model must express all the possible behaviors of the system, the formulas depict required properties of such behaviors. Synchronous Mealy machines are well suited to express these behaviors and they are relevant models to apply model checking techniques.

The properties may be formalized as formulas of a formal logic interpreted over automata. A popular logic is CTL* (*computation tree logic* see [14]). It contains universal and existential quantification over model paths, as well as temporal operators expressing that a property holds in the next state, or in every state (safety properties), or in some state (liveness properties). Nowadays, a lot of tools [4, 10, 19] check CTL* properties against automata models. The logic is interpreted over *Kripke structures* (see 3.2.1) in order to express model checking algorithms and satisfaction of a state formula is defined in a natural inductive way (see [14] for complete definitions). A Mealy machine can be mapped to a Kripke structure, which is also a state machine.

3.2.1 Verification context

In this section, we formally introduce the temporal logic we consider and its Kripke structure model.

Kripke Structures

Kripke structures are verification models against which model-checking algorithms are defined.

A *Kripke structure* K is a tuple: $K =< Q, Q_0, A, R, L >$ where:

1. Q is a finite set of states
2. $Q_0 \subseteq Q$ is the set of initial states
3. A is a finite set of atomic propositions

[5] it is a weak alarm since lying posture is not dangerous in all contexts.

4. $R \subseteq Q \times Q$ is a transition relation that must be total: for every state $q \in Q$, there is a state q' such that $R(q, q')$
5. $L : S \mapsto 2^A$ is a labeling function that labels each state by the set of atomic propositions true in that state.

Let K be a Kripke structure, a *path* in K is an infinite sequence of states: $\pi = q_0, q_1, q_2, ...$ such that $\forall i \in \mathbb{N}, R(q_i, q_{i+1})$. Moreover, $\pi^n = q_n, q_{n+1}, ...$

From Mealy Machine to Kripke structures

Let $M =< Q, q^{init}, I, O, \mathcal{T}, \lambda >$ be a Mealy machine, the Kripke structure $\mathcal{K}(M)$ associated with M is defined as follows: $\mathcal{K}(M) =< Q^K, Q_0^K, A^K, R^K, L^K >$ where

1. $Q^K \subseteq Q \times 2^{A^K} : Q^K = \{(q, v) | \exists (q \xrightarrow{i/o} q' \in \mathcal{T} \text{ and } \{i\} \cup o = v\} \cup \{(q, \emptyset) | q \in Q\}$.
2. $Q_0^K = (\{q^{init}\} \times 2^{A^K}) \cap Q^K$
3. $A^K = I^B \cup O_\epsilon$
4. $L^K(s, v) = v$
5. $((q, v), (q', v')) \in R^K$ iff $\exists q \xrightarrow{i/o} q'$ and $v = \{i\} \cup o$ and $(q', v') \in Q^K$ and $((q, \emptyset), (q, \emptyset)) \in R^K$ for $q \in Q$.

3.2.2 Properties Definition

The logic ($\forall CTL^*$) we consider to express properties is a formal language where assertions related to behavior are easily expressed. It is based on first-order logic but, in order to be efficient when deciding whether a formula is true, the existential path quantifier has been eliminated. It offers temporal operators that make it possible to express properties holding for a given state, for the next state (operator **X**), eventually for a future state (**F**), for all future states (**G**), or that a property remains true until some condition (**U**). We can also express that a property holds for all the paths starting in a given state ($\forall$).

Formally, the logic $\forall CTL^*$ we consider is the set of state formulas defined as follows:

- The constants *true* and *false* are *state* formulas.
- If $p \in A$, p and $\neg p$ are a state formulas (A being the alphabet of the Kripke structure we consider).
- If ψ and ϕ are *state* formulas, then $\psi \vee \phi$ and $\psi \wedge \phi$ are *state* formulas.
- If ϕ is a *path* formula then $\forall(\phi)$ is a *state* formula.
- If ϕ is a *state* formula then ϕ is also a *path* formula.
- If ψ and ϕ are *path* formulas, then $\psi \vee \phi$ and $\psi \wedge \phi$ are *path* formulas.
- If ψ and ϕ are *path* formulas, then so are : $\mathbf{X}\phi$; ϕ **U** ψ; $\mathbf{F}\phi$ and $\mathbf{G}\phi$.

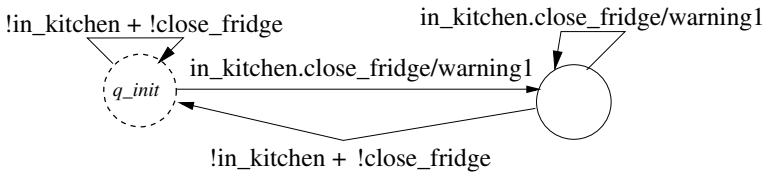

Figure 3: Mealy machine representing the synchronous monitor from camera (called *camera*). The *camera* monitor is a Mealy machine with $\{in\_kitchen, close\_fridge\}$ as input set, $\{warning_1\}$ as output set. When both of its inputs are present, $warning_1$ is emitted.

Satisfaction of formulas

Now, we introduce the notion of "satisfaction of a formula". Given a Kripke structure K ($K =< Q, Q_0, A, R, L >$), the satisfaction of a state formula (ϕ) by a state q of K (denoted $q \models \phi$) or of a path formula ψ by a path π (denoted $\pi \models \psi$) is inductively defined as follows:

- $q \models true$, $q \not\models false$, $q \models p$ iff $p \in L(q)$ and $q \models \neg p$ iff $p \notin L(q)$.
- $q \models \psi \vee \phi$ iff $q \models \psi$ or $q \models \phi$, $q \models \psi \wedge \phi$ iff $q \models \psi$ and $q \models \phi$.
- $q \models \forall(\phi)$ iff for every path π starting at q, $\pi \models \phi$.
- $\pi \models \phi$ where ϕ is a state formula, iff the first state of π satisfies ϕ.
- $\pi \models \psi \vee \phi$ iff $\pi \models \psi$ or $\pi \models \phi$, $\pi \models \psi \wedge \phi$ iff $\pi \models \psi$ and $\pi \models \phi$.
- If ψ and ϕ are path formulas:
  - $\pi \models \mathbf{X}\phi$ iff $\pi^1 \models \phi$.
  - $\pi \models \phi \ \mathbf{U} \ \psi$ iff $\exists n \in \mathbb{N}$ such that $\pi^n \models \psi$ and $\forall i \leq n, \pi^i \models \phi$.
  - $\pi \models \mathbf{F}\phi$ iff $\exists k \in \mathbb{N}$ such that $\pi^k \models \phi$
  - $\pi \models \mathbf{G}\phi$ iff $\forall i \in \mathbb{N}\ \pi^i \models \phi$

Definition 1:
We say that a Kripke structure K satisfies a state formula ψ ($K \models \psi$) if property ψ is true for every initial state of K. This definition is extended to Mealy machines: $M \models \psi$ iff $\mathcal{K}(M) \models \psi$.

In our approach, several synchronous monitors can drive the same proxy component, corresponding to several sub assemblies respectively managing different concerns, all of them related to the critical component. For instance, in the application introduced in section 2, there is another assembly associated with a *camera* sensor also connected to the *warning* entry of the *Alarm* component. Thus, we will define another synchronous monitor telling the behavior of this assembly (see figure 3). The output event $warning_1$ of this *camera* monitor is connected to the *warning* entry of the *Alarm* component. Thus, we must specify how we compose the *posture* monitor and the *camera* monitor to have the expected behavior of the *Alarm* component when it receives both $warning_1$ and $warning_3$.

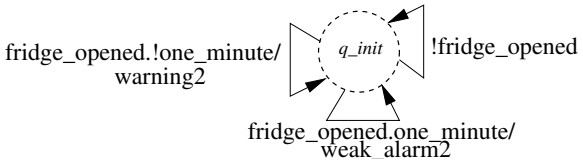

Figure 4: Mealy machine representing the synchronous monitor from fridge (called *fridge*). The fridge monitor reacts to $\{fridge\_opened, one\_minute\}$ input events and sends a $warning2$ event when the fridge door is opened and a $weak\_alarm2$ event when it is opened for more than one minute.

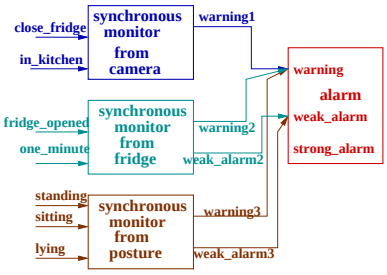

Figure 5: Multiple access to alarm proxy component

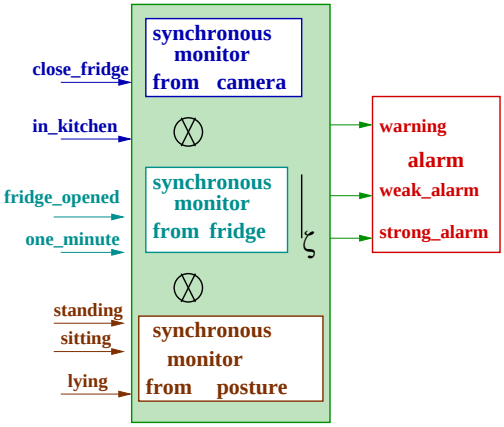

Figure 6: Composition of multiple access to connect the alarm proxy component

4 Synchronous Model Composition

4.1 Multiple Access to Components.

When a critical component has multiple synchronous monitors corresponding to several concern managements in the application, we want to build an only synchronous model component which agrees with all these primitive synchronous monitors and whose output event set is related to the input event set of the critical component by an injection. We continue to rely on our use case (see section 2) to illustrate such a situation.

In this use case, there are three sub-assemblies linked to the critical *Alarm* component. Thus, we introduce three synchronous monitors in this assembly. The first synchronous monitor describes the behavior of *Alarm* component with respect to the sub assembly managing the *camera* device; the second is defined with respect to the sub assembly related to the *door fridge* and *timer* sensors; and the third tells the behavior of *Alarm* when it is related to a sub assembly managing a *posture detection* sensor. *camera* monitor has been described in section 3.2.2 while *posture* monitor was in section 3.1.2. The *fridge* monitor is detailed in figure 4. Then, we get the assembly described in figure 5. We can see that *warning* and *weak_alarm* entries have multiple access. Our method

will replace these three components by a single component : $camera \otimes fridge \otimes posture\ |_\zeta$ (see figure 6).

Now we introduce useful definitions to formally specify our composition operation.

Definition 2:
The synchronous product of two Mealy machines $(M_1 \otimes M_2)$ is defined as follows: assume that
$M_1 =< Q_1, q_1^{init}, I_1, O_1, \mathcal{T}_1, \lambda_1 >$ and $M_2 =< Q_2, q_2^{init}, I_2, O_2, \mathcal{T}_2, \lambda_2 >$,
then $M_1 \otimes M_2 = < Q_1 \times Q_2, (q_1^{init}, q_2^{init}), I_1 \cup I_2, O_1 \cup O_2, \mathcal{T}, \lambda >$ where
$\mathcal{T} = \{((q_1, q_2), (q_1', q_2')) \mid (q_1, q_1') \in \mathcal{T}_1, (q_2, q_2') \in \mathcal{T}_2\}$ and
$\forall (q_1, q_1') \in \mathcal{T}_1 \mid \lambda_1((q_1, q_1'), i_1) = o_1$ and $\forall (q_2, q_2') \in \mathcal{T}_2 \mid \lambda_2((q_2, q_2'), i_2) = o_2$
then $\lambda(((q_1, q_2), (q_1', q_2'), i_1 \cdot i_2) = o_1 \cup o_2$.

The synchronous product considers all the combinations of states, taking into account the simultaneity of events according to our synchronous approach. As already mentioned, in the composition operation we consider only synchronous monitors driving the same proxy component. On one hand, the synchronous product allows to agree with each synchronous monitor. On the other hand, it introduces transitions whose output label carry events belonging to the union of the respective output event sets of $M_1(O_1)$ and $M_2(O_2)$. But we want that the relationship between the output event set of the composition and the input set of the critical component will be at least an injection. Thus, we apply to the synchronous product a "constraint function" defined according to the respective injections $in_1 : O_1 \mapsto I_C$ and $in_2 : O_2 \mapsto I_C$. First, we introduce a new output event set O and an injection $in : O \mapsto I_C$. Second, we define a surjective function $\gamma : O_1 \cup O_2 \cup O_1 \times O_2 \mapsto O_\epsilon$ such that:

1. $\forall o_1 \in O_1$, $\gamma(o_1) = o$ and $in(o) = in_1(o_1)$
2. $\forall o_2 \in O_2$, $\gamma(o_2) = o$ and $in(o) = in_2(o_2)$

From these definitions, a "constraint" function $\zeta : 2^{O_1 \cup O_2} \mapsto 2^O$ is deduced: $\forall o \in 2^{O_1 \cup O_2}$, if $\exists o_1, o_2 \in o$ such that $\gamma(o_1, o_2) \neq \epsilon$ then $\gamma(o_1, o_2) \in \zeta(o)$; else $\gamma(o_1) \in \zeta(o)$ and $\gamma(o_2) \in \zeta(o)$.

This constraint function is applied to the output label sets of of the synchronous product:

Definition 3:
Assume that $M_1 \otimes M_2 = < Q_1 \times Q_2, (q_1^{init}, q_2^{init}), I_1 \cup I_2, O_1 \cup O_2, \mathcal{T}, \lambda >$, then $M_1 \otimes |_\zeta\ M_2 =< Q_1 \times Q_2, (q_1^{init}, q_2^{init}), I_1 \cup I_2, O, \mathcal{T}_\zeta, \lambda_\zeta >$ where $\mathcal{T}_\zeta = \mathcal{T}$ and λ_ζ is defined as follows:
$\lambda_\zeta(((q_1, q_2), (q_1', q_2')), i) = o$ iff $\lambda(((q_1, q_2), (q_1', q_2')), i) = o_1 \cup o_2$ and $\zeta(o_1 \cup o_2) = o$ (assuming that λ is the labeling function of $M_1 \otimes M_2$).

The synchronous product of two Mealy machines yields a Mealy machine. It is a well known result of the synchronous framework. Constraint function application modifies only output event sets of labels and thus our composition operation constructs a Mealy machine.

4.2 Composition and Validation.

Our composition operation allows to solve the multiple access to a given proxy component problem. As previously mentioned, we aim at validating critical component behavior. The result of composition operation is a Mealy machine against which model-checking techniques apply as for any synchronous monitor (see 3.2). Moreover, we also want the preservation of properties under composition: if M_1 verifies an $\forall CTL^*$ formula Φ ($M_1 \models \Phi$) then this latter also holds for a composition where M_1 is part of ($M_1 \otimes |_\zeta M_2 \models \Phi$). To prove such a feature, we show that $\mathcal{K}(M_1)$ can be viewed as an "approximation" of $\mathcal{K}(M_1 \otimes |_\zeta M_2)$.

Definition 4:
Let $K_1 = < Q_1, Q_1^0, A_1, R_1, L_1 >$ and $K_2 = < Q_2, Q_2^0, A_2, R_2, L_2 >$ be two kripke structures and h_a a surjection from A_1 to A_2. We say that K_2 approximates K_1 (denoted $K_1 \sqsubseteq_h K_2$) when

1. It exists a surjection $h : Q_1 \mapsto Q_2$ such that: $h(q_1) = q_2 \Rightarrow \forall a_2 \in L_2(q_2), \exists a_1 \in L_1(q_1)$ and $h_a(a_1) = a_2$.
2. $\forall q_2 \in Q_2^0, \exists q_1 \in Q_1^0$ and $h(q_1) = q_2$; and
3. $\exists q_1, q_1'(h(q_1) = q_2, h(q_1') = q_2'$ and $R_1(q_1, q_1') \Rightarrow R_2(q_2, q_2'))$.

For short, we will denote $M_1 \otimes |_\zeta M_2$ as M_ζ. To benefit from results concerning the preservation of $\forall CTL*$ properties through approximations, we want to show that $\mathcal{K}(M_1)$ is an approximation of $\mathcal{K}(M_\zeta)$.

4.2.1 Approximations for Synchronous Monitors

$\mathcal{K}(M_1) = < \mathcal{K}Q_1, Q_1^0, A_1, L_1, R_1 >$ and $\mathcal{K}(M_\zeta) = < \mathcal{K}Q_\zeta, Q_\zeta^0, A_\zeta, L_\zeta, R_\zeta >$ are built according to the translation operation described in 3.2.1. Our goal is to define a surjective mapping $\hat{h} : \mathcal{K}Q_\zeta \mapsto \mathcal{K}Q_1$ and to show that it agrees with the definition of approximation (definition4).

We first define a surjective mapping $\hat{h}_a$ from A_ζ to A_1. The alphabets of Kripke structures associated with Mealy machines are composed of (1) the Boolean expressions built from inputs and (2) the outputs. Thus, $\hat{h}_a$ must be defined on both.

To ease the definition of $\hat{h}_a$, we start by defining a projection function p_{I_1} from $(I_1 \cup I_2)^B$ to I_1^B. Each element i in $(I_1 \cup I_2)^B$ has a normal form and can be written as $\sum_j \prod_i \omega_i$ where each ω_i is an atom; i.e either an element of $I_1 \cup I_2$ or the negation of an element of $I_1 \cup I_2$. p_{I_1} is defined structurally:

$p_{I_1}(\omega) = \omega$ if ω is an atom in I_1;

$p_{I_1}(\omega) = true$ if ω is an atom in I_2;

$$p_{I_1}(\prod_i \omega_i) = \prod_i p_{I_1}(\omega_i)$$

$$p_{I_1}(\sum_j prod_j) = \sum_j p_{I_1}(prod_j)$$

Lemma 1:
For each element of $(I_1 \cup I_2)^B$ of the form $i_1.i_2$ with i_1 (resp i_2) in I_1^B (resp. I_2^B), $p_{I_1}(i_1.i_2) = i_1$.

Proof. $i_1 = \sum_k \prod_i \omega_i$ where ω_i are atoms in I_1 and $i_2 = \sum_l \prod_j \alpha_j$ where α_j are atoms in I_2.
$p_{I_1}(i_1.i_2) = p_{I_1}(\sum_k \prod_i \omega_i . \sum_l \prod_j \alpha_j) = p_{I_1}(\sum_k \prod_i \omega_i).p_{I_1}(\sum_l \prod_j \alpha_j).$
$p_{I_1}(i_1.i_2) = \sum_k \prod_i p_{I_1}(\omega_i). \sum_l \prod_j p_{I_1}(\alpha_j).$
But $p_{I_1}(\alpha_j) = true$ by definition of the projection function, since α_j is an atom in I_2 and $p_{I_1}(\omega_i) = \omega_i$ since ω_i is an atom in I_1.
Thus $\sum_l \prod_j p_{I_1}(\alpha_j) = true$ and $p_{I_1}(i_1.i_2) = \sum_k \prod_i \omega_i = i_1$. □

According to the translation operation from Mealy machine to Kripke structure, $\mathcal{K}Q_\zeta \subseteq Q_\zeta \times 2^{A_\zeta}$ and $A_\zeta = (I_1 \cup I_2)^B \cup O_\epsilon$.
More precisely, $\mathcal{K}Q_\zeta = \{\{(q_1, q_2), v\} | \exists (q_1, q_2) \xrightarrow{i/o} (q_1', q_2') \in \mathcal{T}_\zeta \text{ and } v = \{i\} \cup o\} \cup \{\{(q_1, q_2), \emptyset\} | (q_1, q_2) \in Q_\zeta\}$.
Similarly, $\mathcal{K}Q_1 = \{\{q_1, v_1\} | \exists q_1 \xrightarrow{i_1/o_1} q_1' \in \mathcal{T}_1 \text{ and } v = \{i_1\} \cup o_1\}\} \cup \{\{q_1, \emptyset\} | q_1 \in Q_1\}$.

First, we define a surjection $\hat{h}_a A_\zeta \mapsto A_1$ as follows: $\forall i \in (I_1 \cup I_2)^B, \hat{h}_a(i) = p_{I_1}(i)$. $\forall o \in O_\epsilon, \hat{h}_a(o) = o_1$ if $o = \gamma(o_1)$ and $\hat{h}_a(\epsilon) = \epsilon$.

To easily express the surjective mapping from $\mathcal{K}Q_\zeta$ to $\mathcal{K}Q_1$, we introduce a function $p_{O_1} : 2^O \mapsto 2^{O_1}$:
$p_{O_1}(o) = \{o_1 | \exists o_2 \in O_2 \text{ and } \gamma(o_1, o_2) \in o\} \cup \{o_1 |\ \nexists o_2 \in O_2 \text{ and } \gamma(o_1, o_2) \in o \text{ and } \gamma(o_1) \in o\}$ Notice that according to the definition of ζ, $p_{O_1}(\zeta(o_1 \cup o_2)) = o_1$. Indeed, from p_{O_1} definition, $\forall o \in o_1$ either there is $o' \in o_2$ such that $\gamma(o, o') \in \zeta(o_1 \cup o_2)$ then $o \in p_{O_1}(\zeta(o_1 \cup o_2))$; or $\gamma(o) \in \zeta(o_1 \cup o_2)$ and then $o \in p_{O_1}(\zeta(o_1 \cup o_2))$.

Lemma 2:
$\mathcal{K}(M_\zeta) \sqsubseteq_{\hat{h}} \mathcal{K}(M_1)$

Proof. We prove that the three conditions of definition 4 are satisfied.
(1) The mapping $\hat{h} : \mathcal{K}Q_\zeta \mapsto \mathcal{K}Q_1$ is defined as follows:
$\hat{h}((q_1, q_2), v) = (q_1, v_1)$ with $v_1 = \{p_{I_1}(i) | i \in 2^{(I_1 \cup I_2)^B} \cap v\} \cup p_{O_1}(v \cap O_\epsilon)$; $\hat{h}$ verifies the required property for surjection: if $\hat{h}((q_1, q_2), v) = (q_1, v_1)$ then $\forall a_1 \in L_1(q_1, v_1), \exists a_\zeta \in L_\zeta(((q_1, q_2), v)$ such that $\hat{h}_a(a_\zeta) = a_1$. By definition, $L_1(q_1, v_1) = v_1$. If $a_1 \in v_1$, then either $a_1 = i_1 \in I_1^B$ or $a_1 = o_1 \subseteq O_1$. Assume $a_1 = i_1$, then by definition of $\hat{h}$, there is $i \in (I_1 \cup I_2)^B$ such that $p_{I_1}(i) = i_1$ and by definition also $\hat{h_a}(i) = i_1$ and $i \in v$ thus $i \in L_\zeta(((q_1, q_2), v)$. Otherwise, $a_1 \in o_1 \subseteq O_1$ and from the definition of $\hat{h}$, $a_1 \in p_{O_1}(v \cap O_\epsilon)$. Thus, from the definition of p_{O_1}, either there is $a_2 \in O_2$ such that $\gamma(a_1, a_2) \in v \cap O_\epsilon$ and then $\hat{h}_a(\gamma(a_1, a_2) = a_1$ either there is not such a_2 and then $\gamma(a_1) \in v \cap O_\epsilon$ and $\hat{h}_a(\gamma(a_1) = a_1$.

(2) $\forall\{q_1, v_1\} \in Q_1^0, \exists\{(q_1, q_2), v\} \in Q_\zeta^0$ and $\hat{h}(\{(q_1, q_2), v\}) = \{q_1, v_1\}$. But $\{q_1, v_1\} \in Q_1^0$ means that $q_1 = q_1^{init}$ and $\exists q_1^{init} \xrightarrow{i_1/o_1} q_1' \in \mathcal{T}_1$ and $v_1 = \{i_1\} \cup o_1$.

1. If there is a transition $q_2^{init} \xrightarrow{i_2/o_2} q_2' \in \mathcal{T}_2$, then by construction there is a transition $(q_1^{init}, q_2^{init}) \xrightarrow{i_1.i_2/\zeta(o_1 \cup o_2)} (q_1', q_2')$ in $\mathcal{T}_\zeta$.
   Thus, $\{(q_1^{init}, q_2^{init}), \{i_1.i_2\} \cup \zeta(o_1 \cup o_2)$ is a state of $\mathcal{K}Q_\zeta$ and an initial state in Q_ζ^0.
   By definition, $\hat{h}(\{(q_1^{init}, q_2^{init}), \{i_1.i_2\} \cup \zeta(o_1 \cup o_2) = (q_1^{init}, \{p_{I_1}(i_1.i_2)\} \cup p_{O_1}(\zeta(o_1 \cup o_2))$. According to lemma 1, $p_{I_1}(i_1.i_2) = i_1$ and we defined p_{O_1} such that $p_{O_1}(\zeta(o_1 \cup o_2)) = o_1$. Thus, $\hat{h}(\{(q_1^{init}, q_2^{init}), \{i_1.i_2\} \cup \zeta(o_1 \cup o_2)$ $= (q_1^{init}, v_1)$.

2. If there is no transition $q_2^{init} \xrightarrow{i_2/o_2} q_2' \in \mathcal{T}_2$, then there is a transition $(q_1^{init}, q_2^{init}) \xrightarrow{i_1/\zeta(o_1)} (q_1', q_2^{init})$ in $\mathcal{T}_\zeta$. In this last case, the result is obvious.

(3) Consider two states in $\mathcal{K}Q_\zeta$, $\{(q_1, q_2), v\}$ and $\{(q_1', q_2'), v'\}$ belonging to the transition relation of $\mathcal{K}(M_\zeta)$ and $\hat{h}(\{(q_1, q_2), v\} = (q_1, v_1)$ and $\hat{h}(\{(q_1', q_2'), v'\} = (q_1', v_1')$. We want to prove that (q_1, v_1) and (q_1', v_1') belong to the transition relation of $\mathcal{K}(M_1)$. But, there is a transition $(q_1, q_2) \xrightarrow{i/o}$ $(q_1', q'2) \in Q_\zeta$ and $v = \{i\} \cup o$. Once again, that means that $\exists q_1 \xrightarrow{i_1/o_1} q_1' \in \mathcal{T}_1$ and $\exists q_2 \xrightarrow{i_2/o_2} q_2' \in \mathcal{T}_2$ and $i = i_1.i_2$ and $o = \zeta(o_1 \cup o_2)$. Then, $v_1 = \{i_1\} \cup o_1$ and by definition $((q_1, v_1), (q_1', v_1') \in R_1$. □

4.3 Approximation and Property Preservation

Now we make more precise what does mean $\forall CTL*$ properties are preserved through our composition operation. In [13], Clarke and all show that $\forall CTL*$ formulas are preserved for transition system approximations. We use the same method to prove that $\forall CTL*$ formulas are preserved through Kripke structure approximations.

4.3.1 $\forall CTL*$ Property Preservation

Let K_1 and K_2 be two Kripke structures and $h_a : A_1 \mapsto A_2$ a surjective mapping such that there is a surjection h from Q_1 to Q_2 and $K_1 \sqsubseteq_h K_2$. The method consists in (1) defining a translation (τ) from formulas expressing properties in K_2 and formulas expressing properties in K_1 and to prove that if a property ϕ holds for K_2, $\tau(\phi)$ holds for K_1.

Definition 5:
The translation τ between formulas in K_2 and formulas in K_1 is defined as follows:

- $\tau(true) = true$, $\tau(false) = false$;
- $\forall a_2 \in A_2, \tau(a_2) = \bigvee\{a_1 \in A_1$ such that $h_a(a_1) = a_2\}$;
- if ϕ and ψ are state formulas, then $\tau(\phi \vee \psi) = \tau(\phi) \vee \tau(\psi)$ and $\tau(\phi \wedge \psi) = \tau(\phi) \wedge \tau(\psi)$;

- if ϕ is a path formula, then $\tau(\forall\phi) = \forall(\tau(\phi))$;
- if ϕ and ψ are path formulas, then $\tau(\phi \vee \psi) = \tau(\phi) \vee \tau(\psi)$ and $\tau(\phi \wedge \psi) = \tau(\phi) \wedge \tau(\psi)$;
- if ϕ and ψ are path formulas, then $\tau(\mathbf{X}\phi) = \mathbf{X}\tau(\phi)$, $\tau(\phi\ \mathbf{U}\ \psi) = \tau(\phi)\ \mathbf{U}\ \tau(\psi)$, $\tau(\mathbf{F}\phi) = \mathbf{F}\tau(\phi)$ and $\tau(\mathbf{G}\phi) = \mathbf{G}\tau(\phi)$.

We now turn to the preservation result. First, we express a straightforward lemma that says that paths in K_1 are projected in K_2(see [13]).

Lemma 3:
If $\pi = \pi_1, ..., \pi_n, ..$ is a path in K_1, then $h(\pi) = h(\pi_1), ..., h(\pi_n), ..$ is a path in K_2.

Relying on this lemma, we prove the preservation theorem:

Theorem 1:
Let K_1 and K_2 two Kripke structures such that $K_1 \sqsubseteq_h K_2$:

1. for all $\forall CTL*$ state formula ϕ in K_2 and for all state q_1 of K_1, $h(q_1) \models \phi \Rightarrow q_1 \models \tau(\phi)$
2. for all $\forall CTL*$ path formula ϕ in K_2 and for every path π in K_1, $h(\pi) \models \phi \Rightarrow \pi \models \tau(\phi)$

Proof. The proof is an induction on the structure of the formula.

1. if $\phi = true$(resp $false$) the result is obvious;
2. if $\phi \in L_2(h(q_1))$, by definition,$\exists a_1 \in L_1(q_1)$ such that $h_a(a_1) = \phi$. Thus $q_1 \models a_1$ and then $q_1 \models \bigvee\{a_1 \in A_1 \text{ such that } h_a(a_1) = \phi\}$. Thus $q_1 \models \tau(\phi)$;
3. if $\phi = \phi_1 \vee \phi_2$: $h(q_1) \models \phi_1$ or $h(q_1) \models \phi_2$. By induction, we know that $q_1 \models \tau(\phi_1)$ or $q_1 \models \tau(\phi_2)$. Thus $q_1 \models \tau(\phi_1) \vee \tau(\phi_2)$ and $q_1 \models \tau(\phi)$. The proof for $\wedge$ is similar;
4. assume $\phi = \forall\psi$, we want to prove that $q_1 \models \tau(\forall\psi)$. This means that for every path π starting from q_1 $\pi \models \psi$. From lemma 3, we know that $h(\pi)$ is a path in K_2 starting from $h(q_1)$. Since $h(q_1) \models \forall\psi$, $h(\pi) \models \psi$. Applying the induction hypothesis, we deduce that $\pi \models \tau(\psi)$;
5. if ϕ is a state formula and $h(\pi) \models \phi$. If the initial state of π is q_1, then the initial state of $h(\pi)$ is $h(q_1)$. Assume that $h(q_1) \models \phi$, then by induction $q_1 \models \tau(\phi)$ and thus $\pi \models \tau(\phi)$;
6. the proofs for $\vee$ and $\wedge$ of path formulas are similar to case (3);
7. if $h(\pi) \models \mathbf{X}\psi$ then $h(\pi)^1 \models \psi$. By induction, $\pi^1 \models \tau(\psi)$ thus $\pi \models \mathbf{X}\tau(\psi)$ and $\pi \models \tau(\mathbf{X}\psi)$;
8. if $h(\pi) \models \phi\mathbf{U}\psi$, there is $n \in \mathbb{N}$ such that $h(\pi)^n \models \psi$ and $\forall i < n, h(\pi)^i \models \phi$. Using the induction hypothesis, we can infer that $\pi^n \models \tau(\psi)$ and $\forall i < n, \pi^i \models \tau(\phi)$. Thus $\pi \models \phi\mathbf{U}\psi$.
9. if $h(\pi) \models \mathbf{F}\psi$, there is $k \in \mathbb{N}$ such that $h(\pi)^k \models \psi$. By induction, we know that $\pi^k \models \tau(\psi)$ and then $\pi \models \mathbf{F}\tau(\psi) = \tau(\mathbf{F}\psi)$.

10. if $h(\pi) \models \mathbf{G}\psi$, then $\forall i \in \mathbb{N}\ h(\pi)^i \models \psi$. By induction, we know that $\forall i \in \mathbb{N}$ $\pi^i \models \tau(\psi)$ and $\pi \models \mathbf{G}\tau(\psi) = \tau(\mathbf{G}\psi)$.

□

4.3.2 Properties Preservation for Synchronous Monitors

Now , we apply these preservation results to synchronous monitors. We recall that we want to prove that if a Mealy machine M_1 satisfies a $\forall CTL*$ formula, then the property holds also in a composition where M_1 is an argument. To this aim, relying on theorem 1, we will show that if $\mathcal{K}(M_1) \models \phi$, a $\forall CTL*$ property, then $\mathcal{K}(M_\zeta) \models \tau_\zeta(\phi)$, τ_ζ being a translation function from formulas related to $\mathcal{K}(M_1)$ to formulas related to $\mathcal{K}(M_\zeta)$.

Definition 6:
The translation mapping τ_ζ between formulas related to $\mathcal{K}(M_\zeta)$ and those related to $\mathcal{K}(M_1)$ is defined as follows:

- $\tau_\zeta(true) = true$, $\tau_\zeta(false) = false$;
- $\forall a_1 \in A_1, \tau_\zeta(a_1) = \bigvee\{a_\zeta \in A_\zeta \text{ such that } \hat{h}_a(a_1) = a_\zeta\}$;
- if ϕ and ψ are state formulas, then $\tau_\zeta(\phi \vee \psi) = \tau_\zeta(\phi) \vee \tau_\zeta(\psi)$ and $\tau_\zeta(\phi \wedge \psi) = \tau_\zeta(\phi) \wedge \tau_\zeta(\psi)$;
- if ϕ is a path formula, then $\tau_\zeta(\forall\phi) = \forall(\tau_\zeta(\phi))$;
- if ϕ and ψ are path formulas, then $\tau_\zeta(\phi \vee \psi) = \tau_\zeta(\phi) \vee \tau_\zeta(\psi)$ and $\tau_\zeta(\phi \wedge \psi) = \tau_\zeta(\phi) \wedge \tau_\zeta(\psi)$;
- if ϕ and ψ are path formulas, then $\tau_\zeta(\mathbf{X}\phi) = \mathbf{X}\tau_\zeta(\phi)$, $\tau_\zeta(\phi\ \mathbf{U}\ \psi) = \tau_\zeta(\phi)\ \mathbf{U}\ \tau_\zeta(\psi)$, $\tau_\zeta(\mathbf{F}\phi) = \mathbf{F}\tau_\zeta(\phi)$ and $\tau_\zeta(\mathbf{G}\phi) = \mathbf{G}\tau_\zeta(\phi)$

Now we can express the preservation theorem for synchronous monitors:

Corollary 1:
Let M_1 and M_2 be two Mealy machines and ϕ a $\forall CTL*$ formula related to M_1, then $M_1 \models \phi \Rightarrow M_1 \otimes |_\zeta\ M_2 \models \tau_\zeta(\phi)$

Proof. By definition, we say that $M_1 \models \phi$ iff $q_1^{init} \models \phi$ and then iff $(q_1^{init}, v_1) \models \phi$ for each initial states of $\mathcal{K}(M_1)$. In section 4.2.1, we have defined a surjective mapping $\hat{h} : \mathcal{K}Q_\zeta \mapsto \mathcal{K}Q_1$ and we have proved that $\mathcal{K}(M_\zeta) \sqsubseteq_{\hat{h}} \mathcal{K}(M_1)$. Let us consider the state $\{(q_1^{init}, q_2^{init}), v\}$ such that $\hat{h}(\{(q_1^{init}, q_2^{init}), v\}) = (q_1^{init}, v_1)$, we have $\hat{h}(\{(q_1^{init}, q_2^{init}), v\}) \models \phi$ as initial hypothesis. Thus, according to theorem 1, we know that $\{(q_1^{init}, q_2^{init}), v\} \models \tau_\zeta(\phi)$. Hence, $M_\zeta \models \tau_\zeta(\phi)$. □

5 Practical Issues

Relying on this theoretical approach, we improve our *WComp* middleware to support synchronous component design and validation of behaviors for critical components.

5.1 Our Reactive Adaptive Middleware

As already mentioned in the introduction, we propose a middleware approach called WComp taking into account all the principles for ubiquitous computing detailed in section 1.1. For that matter, it federates three main paradigms :

1. *Event-based services architecture* : Our services are event-based. They can communicate between them using event patterns to transmit as soon as possible spontaneous information coming from the physical environment. For example we attach a service to a sensor device that sends regularly new measures to other services. They are generally Web Services for Devices like UPnP or DPWS. We distinguish then two kinds of services : *composite services* which are services whose implementation calls other services in the middleware layers. They are opposed to *basic services* from the infrastructure, whose implementation is self-contained and does not invoke any other services. *Ubiquitous applications are then a graph of interactions between event-based services.*

2. *Lightweight component-based architecture inside composite web services* : A Composite Service is based on an internal lightweight component assembly to manage composition between other event-based web services through proxies components and to design the interface of a new higher-level composite service. A Composite Service corresponds to a dynamic assembly of lightweight WComp components and provides an event-based service interfaces, like explained previously. Internal assembly of components handles the high dynamicity of the model, providing a way to be structurally modified and adapted. It also addresses reactivity, since it uses event-based communications between components. *A composite event-based service is dynamically managed using an internal lightweight components assembly.*

3. *Adaptation paradigm using the original concept named Aspect of Assembly (AA)*: This concept allows to prepare kinds of *independent and crosscutting* schemes of adaptation dealing with separation of concerns, logically mergeable in case of conflicts and applicable to every Composite Web Service of the application, not necessarily known (previously). Aspects provide adaptation to the model, which is structural, since we modify the internal component assembly of composite services, without modifying black boxes components. *Adaptations as a set of AA, are designed to modify event-based web services of the application according to the evolution of the infrastructure (appearance and disappearance of devices in it). They are applied (weaved) to the set of event-based composite web services of the applications at runtime to implement then required reactive adaptation*

Thus our middleware allows to adopt both ways to dynamically design ubiquitous computing applications. The first implements a classical component-based compositional approach, using SLCA, to design higher-level composite web services and then increments the graph of cooperating services for the applications. This approach is well suited to design the applications in a known, common and usual way. We call such a compositional approach *composition for higher-level services.*

The other way uses a compositional approach for adaptation using AA, particularly well-adapted to tune a set of event-based web services in reaction to a particular variation of the context. We call such compositional approach *composition for reactive adaptation*.

5.2 Extending WComp

In section 3, we have shown that the introduction of specific synchronous component is an answer to address the multiple access to critical component problem. These synchronous component we introduce represent the behaviors of critical components as Mealy machines. These latter are models very well suited to perform safety property verification, but they are not convenient to deal with. Due to the synchronous product they can become huge and we must face the famous "state explosion problem". To avoid this drawback, we want to benefit from symbolic representation of Mealy machines. Thus we rely on *synchronous languages* [2].

These languages support functional *concurrency* and they rely on notations that express concurrency in a user-friendly manner. They also offer *simple* formal model that makes formal reasoning tractable. In particular, the semantics for the parallel composition of two processes is clearly defined. Finally, they respect the *synchrony* hypothesis which divides time into discrete instants. Hence in a natural way, synchronous programs progress according to successive atomic reactions. Indeed, Mealy machines are models for these languages and their compilation involves the construction of these formal models. Moreover, synchrony and concurrency imply that the synchronous product defined section 4 is exactly the semantics of parallel operator of synchronous languages.

Then, to apply our approach, we rely on the Lustre [17] *synchronous language* which helps us to define and validate synchronous components. It is a data flow language offering two main advantages: (1) It is a *functional* language no complex side effects. This makes it well adapted to formal verification and safe program transformation, since functional relations over data flows may be seen as time invariant properties. Also, reuse is made easier, which is an interesting feature for reliable programming concerns; (2) it is a *parallel* model, where any sequencing and synchronization depends on data dependencies. Thus, the synchronous product we rely on to perform the composition of synchronous components under constraints is expressed naturally in the language. Moreover, constraint functions can be expressed as equations, thanks to the equational nature of the language.

To perform safety properties validation we rely on the model-checking tool Lesar [9], a symbolic, BDD-based model-checker for Lustre. It is based on the use of *synchronous observers* [8], to describe both the properties to be checked and the assumptions on the program environment under which these properties are intended to hold. An observer of a safety property is a program, taking as inputs the inputs/outputs of the program under verification, and deciding (e.g., by emitting an alarm signal) at each instant whether the property is violated. Running in parallel with the program, an observer of the desired property, and an observer of the assumption made about the environment one has just to check that either the alarm signal is never emitted (property satisfied) or the alarm signal is emitted (assumption violated), which can be done by a simple traversal of the reachable states of the compound program. Hence, using

observer technique allows to express the property in the same language used to design our synchronous components and avoid to express non intuitive temporal logic formulas.

5.3 Use Case Implementation

Now we sketch how we implement the use case described in section 2. In our implementation, a synchronous monitor is expressed as a Lustre program (called a *node*). Hence three Lustre nodes implement respectively the three synchronous monitors of the use case:

```
node camera(in_kitchen,close_fridge:bool) returns(warning1:bool)
let warning1 = in_kitchen and close_fridge;
tel
node fridge(fridge_opened, one_minute: bool)
     returns (warning2, weak_alarm2: bool);
let warning2= fridge_opened and not one_minute;
    weak_alarm2= fridge_opened and one_minute;
tel
node posture(sitting, standing,lying:bool)
     returns(warning3,weak_alarm3:bool)
let  warning3 = (standing or sitting) and not lying;
     weak_alarm3 = not standing and not sitting and lying;
tel
```

Figure 5 shows the assembly weaved to design the application. Let $O_1 = \{warning1\}$, $O_2 = \{weak\_alarm2, warning2\}$ and
$O_3 = \{weak\_alarm3,\ warning3\}$ be the respective output sets of *camera*, *fridge* and *posture* components.
Let $I_A = \{warning, weak\_alarm, strong\_alarm\}$ be the input set of the *Alarm* component.
For each component, we defined an injection $in_i : O_1 \mapsto I_A (i = 1, 2, 3)$:

$$\begin{cases} in_i(warning_i) = warning (i = 1, 2, 3) \\ in_i(weak\_alarm_i) = weak\_alarm (i = 2, 3) \end{cases}$$

We replace these three components by a single component : $camera \otimes fridge \otimes posture\ |_\zeta$, as said in section 4. This composite component is a Mealy machine, which has for input event set the union of the respective input event sets of the *camera*, *fridge* and *posture* components i.e {*close_fridge, in_kitchen, fridge_opened, one_minute, standing, sitting, lying*}. The output set of the composite component we built is $O = \{warning, weak\_alarm, strong\_alarm\}$ and we define an injection $in : O \mapsto I_A$:

$$\begin{cases} in(warning) = warning \\ in(weak\_alarm) = weak\_alarm \end{cases}$$

As *in* is a bijection, we kept the same name for output events in O and input events in I_A, to an easier identification of connections. Now, we must provide a surjective function $\gamma : O_1 \cup O_2 \cup O_3 \cup (O_1 \times O_2 \times O_3) \cup (O_1 \times O_2) \cup (O_2 \times O_3) \cup (O_1 \times O_2) \mapsto O_\epsilon$ which agrees with the respective injections $in_i \mapsto I_A$.

$$\begin{cases} \gamma(warning_i) = warning (i = 1, 2, 3) \\ \gamma(weak\_alarm_i) = weak\_alarm (i = 2, 3) \\ \gamma(weak\_alarm_2, weak\_alarm_3) = strong\_alarm \\ otherwise\ \gamma(o) = \epsilon \end{cases}$$

Then we infer a constraint function $\zeta : 2^{(O_1 \cup O_2 \cup O_3)} \mapsto 2^O$ and we apply it to build the composite component $camera \otimes fridge \otimes posture \mid_\zeta$. The constraint function applies to the output sets borne by transitions of *camera*, *fridge* and *posture* synchronous product. It maps all combinations of $warning_1$, $warning_2$ and $warning_3$ to *warning* event. As soon as either $weak\_alarm_2$ or $weak\_alarm_3$ are emitted, ζ maps the output set to *weak_alarm* and if both of them belong to an output set, then a *strong_alarm* is sent since that means that the door of the fridge is opened for more than one minute and the person kept under watch is lying. Of course, different constraints could be defined. For instance, instead of considering that a $warning_i$ is sufficient to launch a warning, we could consider that the *camera* and *fridge* components must agree and emit respectively $warning_1$ and $warning_2$. This would yield another composition result.

To implement the synchronous monitor performing the composition of *camera*, *fridge* and *posture* synchronous monitors, we rely on the natural synchronous parallelism of Lustre. Indeed in this language, the synchronous product is implicit and we only have to call the respective nodes implementing the components to build their synchronous product. Then, to express the constraint function, we define a set of equations describing the computation of each output of the composition (showing in violet in the following):

```
node alarm_comp (close_fridge, fridge_opened, one_minute, standing,
                 sitting, lying, in_kitchen : bool)
     returns (warning, weak_alarm, strong_alarm : bool)

var warning1, warning2, warning3, weak_alarm2, weak_alarm3 : bool;
let  warning1 = camera(in_kitchen, close_fridge);
    (warning2, weak_alarm2) = fridge(fridge_opened, one_minute);
    (warning3, weak_alarm3) = posture(standing, sitting, lying);

    warning = warning1 or warning2 or warning3 and not weak_alarm2
              and not weak_alarm3;
    weak_alarm = weak_alarm2 xor weak_alarm3;
    strong_alarm = weak_alarm2 and weak_alarm3;
tel
```

Now, we want to verify the *alarm_comp* node behavior before introducing it in the assembly. Thus, we use the observer technique previously described to prove that if the fridge is opened for more than one minute and the person is lying, then a strong alarm is sent. To this aim, we define the following *verif* node. It listens all the entries the *alarm_comp* node listens and it computes a Boolean output prop. Then the model checker Lesar verifies that prop is always true, assuming that *standing*, *sitting* and *lying* are exclusive.

```
node verif (close_fridge, fridge_opened, one_minute, standing,
            sitting, lying, in_kitchen : bool) returns (prop: bool)
var warning, weak_alarm, strong_alarm : bool;
let (warning, weak_alarm, strong_alarm) =
           alarm_comp(close_fridge, fridge_opened, one_minute,
                standing, sitting, lying, in_kitchen);
    assert (not ((standing and lying) or (standing and sitting) or
                (lying and sitting)));

   prop = if (fridge_opened and one_minute and lying) then strong_alarm
          else true;
tel
```

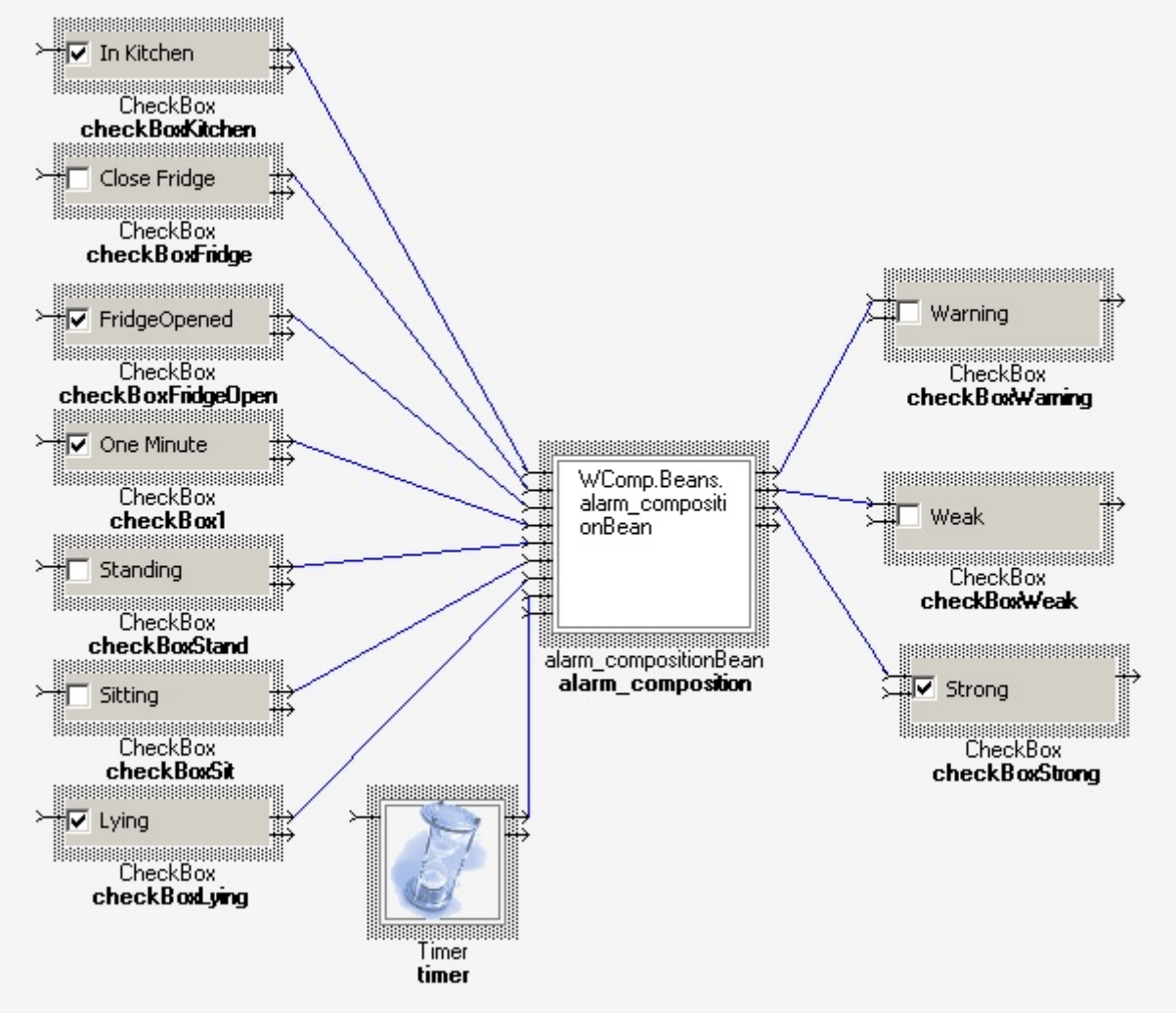

Figure 7: Use case final assembly. The component alarm_composition has been automatically generated from **alarm_comp** Lustre node.

On another hand, we just want to touch on the application of property 1. Assume that with Lesar, we prove that for *fridge* component, the property: $fridge\_opened \Rightarrow warning_2$ holds. Clearly, from the definition of constraints in *alarm_comp*, we have $\tau_\zeta(warning_2) = warning$.
Thus, we can deduce that $fridge\_opened \Rightarrow warning$ also holds in *alarm_comp*.

After this verification, we automatically generated WComp input code for node *comp*. Thus this new component has been automatically weaved in the assembly designing the application in WComp. Figure 7 shows the resulting assembly.

6 Related Works

In this work, we rely on a synchronous modelling to verify the functional correctness under concurrency of component behavior and component assemblies in a reactive and adaptive middleware. Other works address the reliability of middleware. For instance, in [22], the authors propose the TLAM (two-level actor model) approach for specifying and reasoning about components of open distributed systems. They show, using the QoS broker MM architecture, how the TLAM framework can be used to specify and reason about distributed middleware services and their composition. They have also shown how specifications in the TLAM framework can lead to implementations. They proved that the implemented middleware correctly works (provided that middleware services respect a set of constraints) and they planned to rely on a theorem prover to achieve these proofs and automate their method.

However, in the same vein as our approach, some works rely on model-checking techniques to ensure the reliability of middleware solutions. For instance, *PolyORB* is a schizophrenic ("strongly generic") middleware offering several core functions and a Broker design pattern to coordinate them. In [12], Hugues and al, generate Petri nets to model the Broker design pattern of *PolyORB* and use model checking techniques relying on Petri nets models to verify qualitative properties (deadlock, bounds of buffers, appropriate use of critical section,...). We don't use such a modelling because (1) we want to rely on a user-friendly method to describe critical unknown component behaviors; (2) properties we consider don't require Petri nets modelling to be checked. Thus, we prefer to rely on a language allowing to express both component behavior and properties to be checked. The work presented in [3] is close to our approach, but is not applied in the same context. The authors present a compositional reasoning to verify middleware-based Software Architecture. They take advantage of the particular structure of applications due to their middleware-based approach to apply the "assume-guarantee" paradigm in their verification process. In this paradigm the validation of a global property is reduced to the verification of local properties against sub components. We share with us the same verification context, but instead of proving global properties by decomposition, we are interested to verify local critical component or assemblies. Following our philosophy, we want to prove properties on the behavior of small entities and thus only a preservation of property validation through a composition operation is relevant for our approach. In [6], Delaval and all also use a synchronous data flow language complemented with a mechanism to depict component contrats (BZR) to extend a high level component-based model (Fractal) in order to enforce safety properties concerning component interactions during dynamic reconfiguration. Indeed from Fractal specification it is possible to extract a BZR program made of the automata representation of the component behavior and the component contract. Then, using an ad-hoc discrete controller synthesis tool, they generate in a target executive middleware of Fractal (C, Java) an additional validated controller. But, common component-based middleware as WComp do not supply enough information to deduce component behaviors and constracts. Then, we solve the problem of safe reconfiguration in relying on sound composition of user-defined synchronous monitors, which operation preserves component properties already proved. Finally, we want to mention Shin Nakajima [18] work which shows that model-checking techniques are well

suited to verify the reliability of web service flows. He relied on SPIN model-checker [10] to verify the reliability of web service flows expressed as WSFL descriptions. The properties validated are reachability, dead-lock freedom or application specific progress properties.

7 Conclusion and Future Works

The work described in this paper is derived from our experience in providing support for correct assembly of components in an event-based reactive and adaptive middleware. In this latter, we solved the adaptation paradigm using the Aspect of Assembly concept. When using our middleware, a developer benefits from a composition mechanism between different component assemblies to adapt his application to context change. While defining this composition mechanism, we realized the need to formalize and verify the multiple access to a critical component (i.e related to a critical device). The corresponding formalism, the topic of this paper, relies on formal methods. Our approach introduces in a main assembly, a synchronous component for each sub assembly connected with a critical component. This additional component implements a behavioral model of the critical component and model checking techniques apply to verify safety properties about it. Thus, we consider that the critical component is validated. Then we proposed a sound (with respect to our mathematical formalism) composition operation between synchronous components. We proved that this operation preserves already separately verified properties of synchronous components. This operation is an answer to the multiple access to critical components. Our aim is to improve our middleware WComp with a dedicated tool. Currently, we supply a graphical interface to design both critical component behaviors and properties as observers in the synchronous language Lustre (see section 5). Then the validation of properties and the creation of the validated synchronous component is automatic. But, designing with Lustre language is not obvious for any expert user and in the future we aimed at providing a user-friendly interface to express critical component behaviors and properties. This interface will report about violation of properties relying on powerful model checker as *NuSMV* [4] and straightly (without using the Lustre compiler) generate internal code to implement synchronous monitors.

From a theoretical point of view, we aim at improving the efficiency of the composition mechanism. Instead of replacing synchronous components by their composition, we want to supply a composition synchronous monitor listening the output events of the original synchronous monitors and achieving their composition. Then we must prove that this assembly yields to the same result than to perform the composition under constraints of synchronous monitors. On another hand, a hard and long term challenge is to take into account uncertainty in critical component modelling. Indeed, in some applications, some sensor devices could deliver non accurate information. Then, it would be more realistic to study others models in order to model behavior of critical component when they intervene in an assembly listening an uncertain context. Only few approaches in synchronous domain offer this feature and model-checking techniques which consider timed or stochastic automata as models are nowadays non efficient.

References

[1] *ARM '10: Proceedings of the 9th International Workshop on Adaptive and Reflective Middleware*, New York, NY, USA, 2010. ACM.

[2] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Readings in hardware/software co-design*, pages 147–159, 2002.

[3] Mauro Caporuscio, Paola Inverardi, and Patrizio Pelliccione. Compositional verification of middleware-based software architecture descriptions, 2004.

[4] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: an OpenSource Tool for Symbolic Model Checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceeeding CAV*, number 2404 in LNCS, pages 359–364, Copenhagen, Danmark, July 2002. Springer-Verlag. Available from: http://nusmv.irst.itc.it.

[5] E.M. Clarke, E.A. Emerson, and A.P Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.

[6] Gwenaël Delaval and Éric Rutten. Reactive model-based control of reconfiguration in the fractal component-based model. In *13th International Symposium on Component Based Software Engineering (CBSE 2010)*, Prague, Czech Republic, June 2010. Available from: http://pop-art.inrialpes.fr/people/delaval/pub/delaval-cbse10.pdf.

[7] Paul Grace. Dynamic adaptation. In H. Miranda B. Garbinato and L. Rodrigues, editors, *Middleware for Network Eccentric and Mobile Applications*, pages 285–304. Springer, 2009.

[8] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[9] N. Halbwachs and P. Raymond. Validation of synchronous reactive systems: from formal verification to automatic testing. In *ASIAN'99, Asian Computing Science Conference*, Phuket (Thailand), December 1999. LNCS 1742, Springer Verlag.

[10] G.J. Holzmann. The Spin Model Checker. *IEEE Trans. on Software Engineering*, 23:279–295, 1997.

[11] Valerie Issarny, Mauro Caporuscio, and Nikolaos Georgantas. A perspective on the future of middleware-based software engineering. In *2007 Future of Software Engineering*, FOSE '07, pages 244–258, Washington, DC, USA, 2007. IEEE Computer Society. acmid = 1254722. Available from: http://dx.doi.org/10.1109/FOSE.2007.

2, http://dx.doi.org/http://dx.doi.org/10.1109/FOSE.2007.2 doi:http://dx.doi.org/10.1109/FOSE.2007.2.

[12] J.Hugues, L.Pautet, and F.Kordon. Refining middleware functions for verification purpose. In *Proceedings of the Monterey Workshop 2003 (MONTEREY'03)*, pages 79–87, Chicago, IL, USA, September 2003.

[13] E. M. Clarke Jr., O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions om Programming Languages and Systems*, 16(5):1512–1542, 1994.

[14] E. M. Clarke Jr., O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[15] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H.C. Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, 2004. Available from: http://portal.acm.org/citation.cfm?id=1008751.1008762, http://dx.doi.org/10.1109/MC.2004.48 doi:10.1109/MC.2004.48.

[16] G. Mealy. A method to synthesizing sequential circuits. *Bell Systems Technical Journal*, pages 1045–1079, 1955.

[17] F. Lagnier N. Halbwachs and C. Ratel. Programming and verifying critical systems by means of the synchronous data-flow programming language lustre. Special Issue on the Specification and Analysis of Real-Time Systems. IEEE Transactions on Software Engineering, 1992.

[18] S. Nakajima. Verification of web service flows with model-checking techniques. In *CW '02: Proceedings of the First International Symposium on Cyber Worlds (CW'02)*, page 0378, Washington, DC, USA, 2002. IEEE Computer Society.

[19] P. Pettersson and K. Larsen. Uppaal2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, 2000.

[20] M. Satyanarayanan. Fundamental challenges in mobile computing. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 1–7, New York, NY, USA, 1996. ACM. acmid = 248053. Available from: http://doi.acm.org/10.1145/248052.248053, http://dx.doi.org/http://doi.acm.org/10.1145/248052.248053 doi:http://doi.acm.org/10.1145/248052.248053.

[21] J.-Y. Tigli, S. Lavirotte, G. Rey, V. Hourdin, and M. Riveill. Lightweight service oriented architecture for pervasive computing. *IJCSI International Journal of Computer Science Issues*, 4(1), September 2009. ISSN (Online): 1694-0784, ISSN (Print): 1694-0814 paper.

[22] Nalini Venkatasubramanian, Carolyn Talcott, and Gul A. Agha. A formal model for reasoning about adaptive qos-enabled middleware. *ACM Trans. Softw. Eng. Methodol.*, 13(1):86–147, 2004. doi:http://doi.acm.org/10.1145/1005561.1005564.

[23] M. Weiser. The computer for the twenty-first century. *Scientific American Ubicomp Paper*, 265:94–104, September 1991.

Centre de recherche INRIA Sophia Antipolis – Méditerranée
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399